8

- Network programming
  - Addressing
  - Using raw sockets
    - TCP
    - UDP
    - Multicast
  - Using RPC via RMI
  - Using web services

Addressing

- In distributed systems, efficiently **addressing** the right host is critical
- Many **request brokers** perform some form of lookup procedure
  - e.g., DNS lookup, RMI registry, CORBA naming service, Universal Plug&Play (UpnP)
  - “hidden” costs!
- In most cases:
  - IP addresses (independent from the media)
  - Media-specific addresses (e.g., Bluetooth)

IP Addressing

- Traditional IP (v4) addressing
  - 32 bits, 4 groups of 8 bit each
  - Notation: decimal-dotted, 131.114.6.36
- Extended IPv6 addressing
  - 128 bits, 8 groups of 16 bit each
  - Notation: hex-:, 2000:fdb8::1:00ab:853c:39a1
    - “0”s can be omitted or skipped
- Private networks
  - Addresses can be assigned at will, as long as **externally visible** addresses are registered

IP Addressing

- Each **host** can have multiple **interfaces**
  - Most often, 1 interface = 1 network card or chip
- Each interface has one **IP address**
  - Configurable
- Each interface has one **physical address**
  - Usually not configurable – MAC address
  - Typical, but not strictly needed
- Each IP address can be referenced by multiple **domain names**
  - Configurable
  - A domain name can resolve into multiple IPs

IP addressing

- A particular process running on a given host is identified by a **port number**
  - port numbers are 16-bit unsigned integers
  - 0-1023 reserved for OS use
  - 1024-65535 available for user applications
  - It is the process' responsibility to bind to a particular port number
  - Port numbers can be agreed-upon (e.g., 80 for HTTP) or negotiated or communicated across servers

Direct addressing in a PN

- A system could use a directly-encoded address space
- Examples
  - Grid topology
    - 64 hosts, organized in a 8x8 square grid
    - Host at coordinates *x*,*y* in the grid has address 192.168.0.(*x*<<3 | *y*)
  - Ring topology
    - 20 hosts, organized in a ring
    - Host *x* has neighbours (*x*-1)%20 and (*x*+1)%20
    - Each host is at IP address 192.168.1.*x*

Direct addressing in a PN

- However, direct addressing suffers severe limitations
  - It is strictly linked to a fixed topology
    - Low scalability, low flexibility
    - Harder to maintain – need exactly that particular addresses assigned to work
  - It is global in nature
    - No dynamic discovery
  - It only works in local networks
    - Hard to distribute geographically

Domain name system

- A distributed application could use DNS names to refer to other hosts
- In favour
  - Easily extensible, dynamic
- Against
  - Lookup costs
    - Reduced by caching: the “working set” of a single host is typically small
  - Administration costs
    - Need to run your own DNS server

IP addressing in Java

- The class InetAddress represents network addresses
- InetAddress has static methods to
  - Map names to addresses
  - Map addresses to names
  - Discover your own address
  - Check various properties of addresses
  - Compare addresses for equality

InetAddress

- `public static InetAddress[] getAllByName(String hostname) throws UnknownHostException`
- `public static InetAddress getLocalHost() throws UnknownHostException`
- `public String getHostName()`
- `public byte[] getAddress()`
- `public String getHostAddress()`

InetAddress

- InetAddress uses an embedded **cache** to avoid unnecessary DNS lookups
- For “a.b.c.d” numeric addresses, no lookup is performed
- For “f.q.n” names, both positive lookups and negative lookups are cached
  - Positive lookups are cached **forever**
  - Negative lookups are cached for 10 seconds
    - Configurable through properties `networkaddress.cache.ttl` and `networkaddress.cache.negative.ttl`

InetAddress - example

```
private static String lookup(String host) {
   InetAddress node;
   try {
      node = InetAddress.getByName(host);
      System.out.println(node);
      if (isHostName(host)) // checks [0-9.]+
           return node.getHostAddress();
      else
         return node.getHostName();
   } catch (UnknownHostException e)
      return "non ho trovato l’host";
}
```

Sockets

- A socket is the data structure used to manage the state of a network connection in the IP
- The operating system provides calls to create, use, destroy sockets
- These are typically mirrored in a language-specific library
- In Java:
  - class Socket
  - class ServerSocket
  - class DatagramSocket

TCP: Creating Sockets

- Class java.net.Socket
  - Represents a (client) Socket
- Constructors: create and connect a socket
  - `public Socket(InetAddress host, int port) throws IOException`
  - `public Socket(String host, int port) throws UnknownHostException, IOException`
  - `public Socket(String host, int port, InetAddress localAddress, int localPort) throws ...`

TCP: Reading/writing from/to Sockets

- Each (client) Socket is associated to two streams, one for input, one for output
- `public InputStream getInputStream( ) throws IOException`
- `public OutputStream getOutputStream( ) throws IOException`
- You can read from and write to those streams through normal Java I/O methods
- Beware of **buffering**!

TCP: read/write example

- Setting up the connection to a server

```
private Socket sock;
private OutputStream os;
private InputStream is;

public TestClient(String host, int port)
throws IOException {
   sock=new Socket(host,port);
   os=sock.getOutputStream();
   is=sock.getInputStream();
}
…
```

TCP: read/write example

- Send/recv messages (1 byte here)

```
private void sendC(byte c) throws IOException {
     byte[] buf= new byte[1];
     buf[0]=c;
     os.write(buf);
}

private byte recvC() throws IOException {
   byte[] cmd=new byte[1];
   int r=is.read(cmd);
   if (r==-1) throw new IOException("…");
   return cmd[0];
}
```

TCP: Closing

- A few notable methods
  - `sock.close()`
    - Ensures the socket is closed, partially-filled buffers sent out, resources freed and re-usable
  - `sock.shutdownInput()`
    `sock.shutdownOutput()`
    - Asymmetrical close – no further input/output possible
  - `flush()`, `close()` on input and output stream
    - Usual semantics
  - `sock.setSoKeepAlive(true)`
    - Sets the SO_KEEPALIVE flag on the socket
    - Automatic periodic “ping”; if no answer, socket is reset

TCP: Closing

- Socket behaviour on close
  - `Sock.setSoLinger(boolean linger, int time)`
  - linger=false (default)
    - send buffer is sent out; recv buffer is discarded
    - close() is asynchronous; errors in sending are not reported
  - linger=true, time=0
    - Both send and recv buffer are discarded
    - close() is asynchronous
  - linger=true, time>0
    - Send buffer is sent out; recv buffer is discarded
    - Close() is synchronous; the call blocks until the data have been received, or timeout time has expired

TCP: Controlling the buffering

- Sock.getReceiveBufferSize(), sock.getSendBufferSize()
- Sock.setReceiveBufferSize(), sock.setSendBufferSize()
- Sock.setTcpNoDelay(boolean enabled)
  - enabled=true → enables Neagle's algorithm (coalescing)
  - enabled=false → disables Neagle's algorithm (transmit immediately)

TCP: Creating server sockets

- Class java.net.ServerSocket
  - Represents a server socket
- Constructor
  - **`public ServerSocket(int port) throws BindException, IOException`**
    - Creates and binds a server socket that is listening (waiting for connections) on the given port
    - Several variants exist for fine-tuning of various parameters
- As soon as the server socket is created, clients can start connecting to the server

TCP: Handling incoming clients

- The socket method accept() suspends the calling thread until a connection request from a client arrives
- At that point, accept() returns another (different) socket, which can be used for communicating with that particular client
- Typical strategies
  - Create a new server thread to handle it
  - Handle the client immediately, then go back to accept()

TCP server: example

- Creates a server

```
ServerSocket ssocket;

public Server(int port) throws IOException {
   ssocket=new ServerSocket(port);
}
```

- Handles a client connection (dedicated thread)

```
while (!done) {
   try {
      Socket s=ssocket.accept();
      new ServiceThread(s).start();
   } catch (IOException e) { … }
}
```

UDP: Creating a socket

- Class java.net.DatagramSocket
- Constructors
  - `public DatagramSocket() throws SocketException`
    - Creates a datagram socket and binds it to a system-selected unspecified free port (usually: client role)
  - `public DatagramSocket(int port) throws SocketException`
    - Creates a datagram socket and binds it to the given port (usually: server role)

UDP: packets

- A packet of data (i.e., a single message) is represented by a java.net.DatagramPacket
- Double role
  - Create and fill in a DP, then give to to a DS for sending out the message inside the packet
  - Create an empty DP, pass it to a DS to store a received message inside the packet
- Can reuse packets or buffers for efficiency
  - Essentially, DP = char array + length + offset
  - Also includes info about partners' IPs & ports

UDP: packets

- Two important observations
  - TCP is a **connection-oriented** protocol
    - Identity of the partners is fixed once and for all upon establishing the connection between them
  - UDP is a **connection-less** protocol
    - Each and every packet must specify its sender and its receiver (IPs & ports)

  - TCP is a **stream-oriented** protocol
    - All traffic is a sequence of bytes, broken at arbitrary boundaries
  - UDP is a **packet-oriented** protocol
    - Each message is sent individually (no guarantees!)

UDP: packets

- Creating a packet
  - `DatagramPacket(byte[] buf, int offset, int length, InetAddress address, int port)`
  - Creates a packet addressed to *address*,*port* and holding the slice of the array *buf* starting at *offset* and of length *length* as payload
  - Variants with fewer parameters exist
  - Setter methods exists to set or change individual parameters
  - Getter methods for inspecting data on an incoming packet

UDP: sending and receiving

- **`DatagramSocket.send(DatagramPacket dp)`**
  - Will send the payload of the packet to the address/port specified in the packet
- **`DatagramSocket.receive(DatagramPacket dp)`**
  - Will suspend until a packet is received
  - Then, it will copy the payload (and the sender info) into the given dp and return
- As for TCP, the programmer can specify the size of the send and receive buffers
  - Within reason - “extra” packets are discarded

UDP: example

- Sending UDP packets

```
InetAddress ia = InetAddress.getByName(host);

DatagramSocket ds = new DatagramSocket();

byte[] data = new byte[20];

/* fill data as needed */

DatagramPacket dp = new DatagramPacket(data,
data.length, ia, port);

ds.send(dp);
```

UDP: example

- Receiving UDP packets

```
DatagramSocket ds = new DatagramSocket(port);

byte[] buffer = new byte[200];

DatagramPacket dp = new DatagramPacket(buffer,
buffer.length);

ds.receive(dp);

byte[] data = dp.getData();

int len = dp.getLength();

/* use the data as needed */
```

Encoding of messages

- All the TCP and UDP messaging is done as sequences of bytes (chars)
- Various standard methods can be used to encode arbitrary messages as sequences of bytes
  - Using ASCII strings
  - Using a ByteArrayInputStream / OutputStream +
    - Using DataInputStream / OutputStream (per-item)
    - Using ObjectInputStream / OutputStream (serialization)
  - Composing messages “by hand”
    - Possibly using bitwise operators

Multicast

- **Multicast** = one-to-many
  - A packet sent by an host is received **at the same time** by multiple other hosts on the same net
  - Joining a **multicast group** is voluntary
- **Broadcast** = one-to-all
  - A packet sent by an host is received **at the same time** by all other hosts on the same net
  - Often used for zero-conf services
- No one-to-one connection possible
  - Hence, all based on UDP

Multicast: group address

- All IP addresses in the range 224.0.0.0 – 239.255.255.255 are reserved for **multicast groups**
- Two reserved addresses
  - 224.0.0.1 = all hosts on the subnet (try ping!)
  - 224.0.0.2 = all routers on the subnet (usually disabled by network administrators)
- Other addresses may be reserved at IANA
  - e.g., 224.0.1.1 = NTP network time service

Multicast theory of operation

- An host can **join** one or more multicast group
- All datagram packets sent to a multicast group address is delivered to **all** hosts that have joined the group
- An host can **leave** a group at any time
- The **port** a packet is sent towards is not significant anymore
  - But the receiver can still retrieve it from the packet
  - Might be used to “tag” different types of traffic

MulticastSocket: example

- A multicast receiver

```
InetAddress group = InetAddress.getByName(gr);
if (!group.isMulticastAddress()){
   throw new IllegalArgumentException();
}
MulticastSocket ms = new MulticastSocket(port);
ms.joinGroup(group);
DatagramPacket dp = new DatagramPacket(new byte[K],K);
ms.receive(dp);
byte[] data=dp.getData();
int len = dp.getLength();
/* process data */
ms.leaveGroup(group);
```

Subclass of DatagramSocket

MulticastSocket: example

- A multicast sender

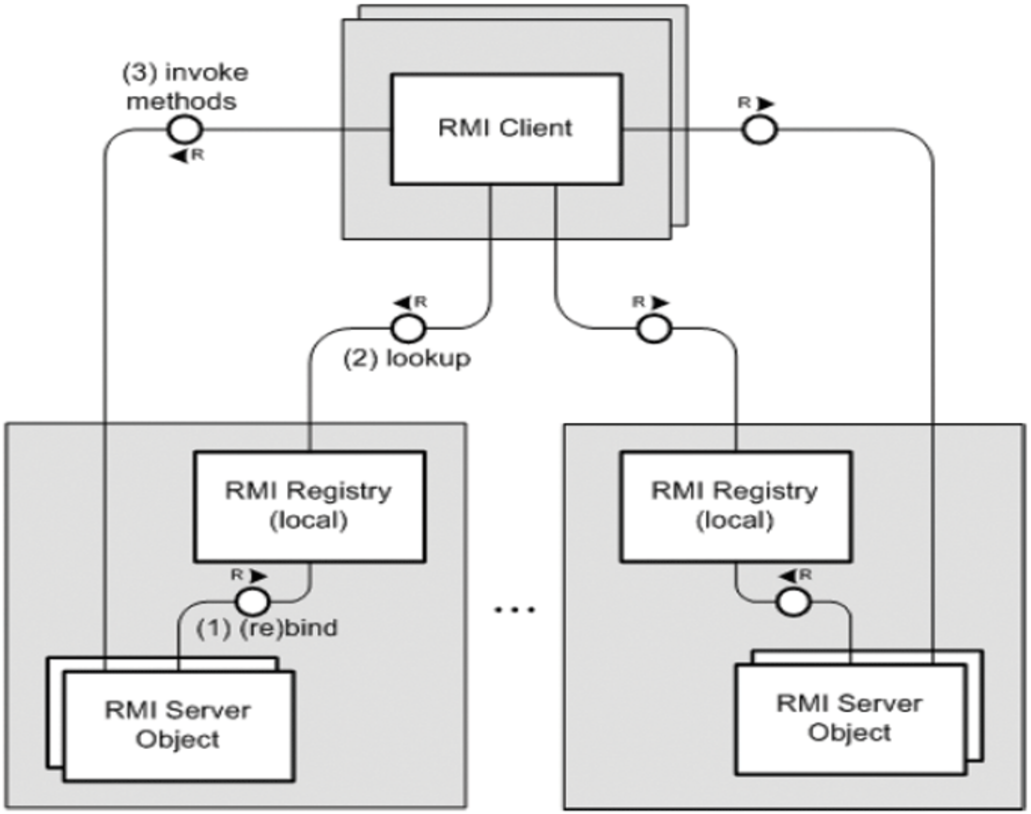
```
InetAddress group = InetAddress.getByName(gr);
if (!group.isMulticastAddress()){
   throw new IllegalArgumentException();
}
MulticastSocket ms = new MulticastSocket();
/* prepare data in d */
DatagramPacket dp = new DatagramPacket(d,d.length);
ms.setTimeToLeave(1);
ms.send(dp);
```

Maximum number of hops
1 = cannot leave local netword

RMI: Remote Method Invocation

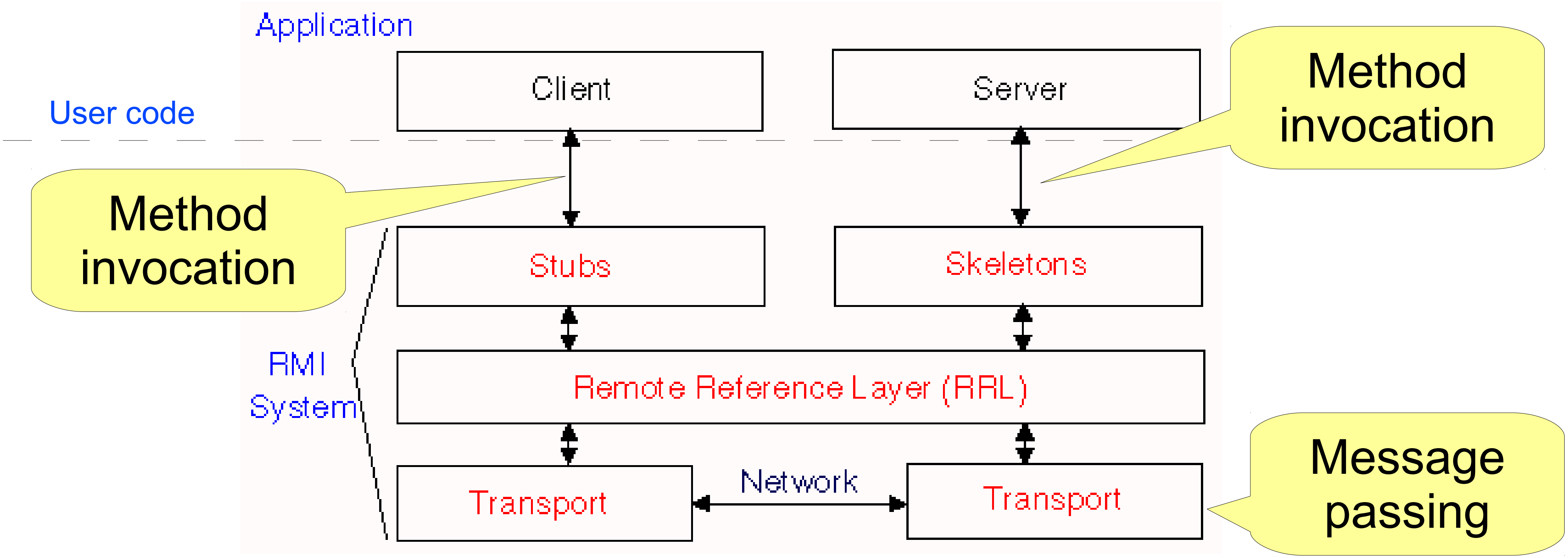
- **RMI** is an infrastructure for causing the execution of methods of objects that reside on a different host
- The caller invokes the operation “as if” it was calling a method
- The callee receives the invocation “as if” it was simply called
- Under the hood, complex marshalling and object serialization is used to provide transparency

RMI overview



- RMI servers export services through a registry
- RMI clients can query the registry to discover services
- Once bound, clients can invoke methods of the server objects

RMI overview



Method invocation

Method invocation

Message passing

- Calls are **routed** through a Stub (client side) and a Skeleton (server side)
- **Remote References** managed by a RRL

RMI: server-side API

- The server object **must** implement java.rmi.Remote
  - Just a marker interface
- All server methods **must** declare that they might throw java.rmi.RemoteException
- The server object **must**
  - extend UnicastRemoteObject **or**
  - call UnicastRemoteObject.exportObject(srvrObj);
- The stub class can be created by running the RMI-Compiler rmic

RMI: server-side API

- To ensure that the server object is available from a registry
  - A registry server **must** be running on the host
    - On Linux: rmiregistry &
    - On Windows: start rmiregistry
  - A symbolic name must be bound to the server object
    - `void Naming.bind(String name, Remote obj)`
    - `void Naming.rebind(String name, Remote obj)`

RMI Server example

- The server object interface

```
import java.rmi.*;
public interface EchoInt extends Remote {
   String getEcho(String echo) throws RemoteException;
}
```

- The server object implementation

```
public class Server implements EchoInt {
   public Server() { ; }
   public String getEcho(String echo)  {
      return echo ;
   }
}
```

The implementation can have private and public methods at will, but only those declared in the Remote interface can be accessed remotely

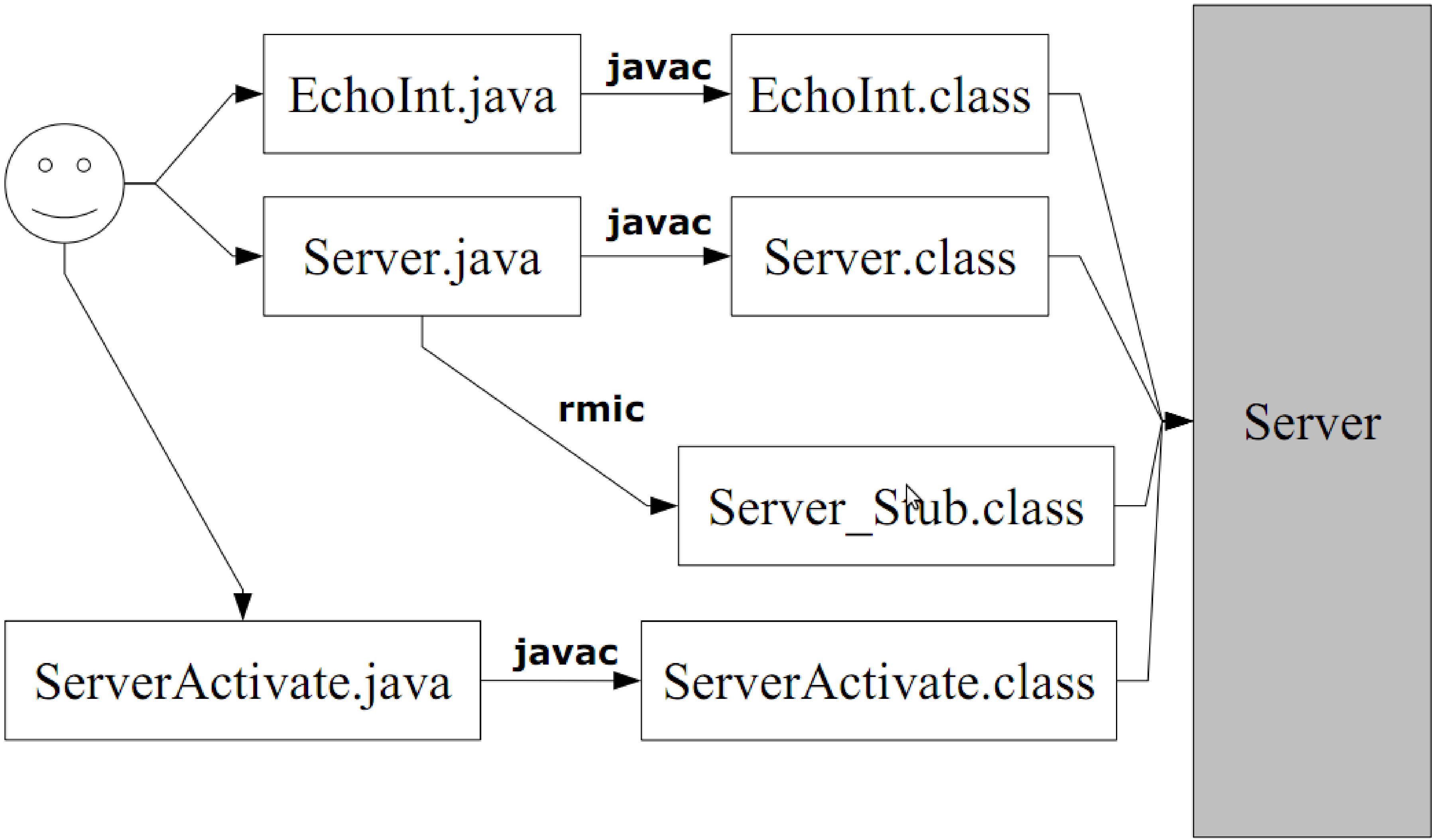
RMI Server example

- Registration of the remote object

```
import java.rmi.registry.*
import java.rmi.server.*;

public class ServerActivate {
   public static void main(String args[]) {
      try {
         Server obj = new Server();
         EchoInt stub =
           (EchoInt)UnicastRemoteObject.exportObject(obj);
         Registry registry = LocateRegistry.getRegistry();
         registry.bind("Echo", stub);
     } catch (Exception e)  { … }
    }
}
```

RMI Server example



EchoInt.java

javac

EchoInt.class

Server.java

javac

Server.class

rmic

Server_Stub.class

ServerActivate.java

javac

ServerActivate.class

Server

RMI: client-side API

- The client locates a registry
  - Often – but not necessarily – the registry is located on the server host
- The client obtains a reference to the remote object from the registry
  - The client **must** have the interface .class!
- Calls to the methods of the obtained object will beahave “as if” the object was local
  - Tons of caveats apply
    - In particular: arguments, results, exceptions are **serialized**!
    - Remote call can fail due to network problems

RMI Client example

- Invoking remote methods

```
try {
   Registry registry = LocateRegistry.getRegistry(host);
   EchoInt stub = (EchoInt) registry.lookup("Echo");
   String response = stub.getEcho(next);
   System.out.println("response: " + response);
} catch (Exception e) { … }
```

- Notice how the client “knows” the remote interface, but **not the implementation**
- On the server, each client is a **different thread** executing the method code
  - Synchronization might be necessary

Web Services

- Similar to other object distribution infrastructure
  - e.g., RMI or Corba
- Remote operations are invoked through SOAP messages on top of HTTP
  - SOAP: an XML-based object serialization protocol
  - Services are “hosted” by a web server
- Rich semantics and types
  - Services are self-described, no need to have IDL
- Slow – useful for heavy-weigth transactions

Web services in Java: server

- Modern tools use Java **annotations**

```
package server;
import javax.jws.WebService;
import javax.jws.WebService;
import javax.xml.ws.Endpoint;

@WebService public class Calculator {

   @WebMethod public int add(int a, int b) { return a+b; }

   public static void main(String[] args){
      Calculator calc = new Calculator();
      Endpoint endpoint =
     Endpoint.publish("http://localhost:8080/calc", calc);
   }
}
```

Web services in Java: deploy

- The Java compiler will recognize the special annotations, and generate
  - A WSDL file describing the web service
  - A .class file containing the compiled bytecode for Calculator
  - Various stubs for additional “hidden” classes
- The interface for the web service can be inspected with a browser at http://localhost:8080/calculator?wsdl
  - Thanks to an internal lightweight web server

Web services in Java: deploy

- The Java compiler will recognize the special annot
  - A W
  - A .c
    Cal
  - Var
- The in
  inspe
  http://
  - Tha

Need proper tools installed!

Some options:

Java Enterprise edition (Java EE)
Java Web Services Developement Pack (JWSDK)
GlassFish
Apache Geronimo
JBoss

or any equivalent web app server

Web services in Java: client

- Client-side tool can generate the stubs
  - wsimport -p client http://localhost:8080/calculator?wsdl
  - Will generate a number of classes
    - Of interest: *Calculator* and *CalculatorService*
- Client code can simply invoke generate classes

```
package client;
class CalculatorApp {
   public static void main(String args[]){
      CalculatorService serv = new CalculatorService();
      Calculator calc = serv.getCalculatorPort();
      int result = calc.add(10, 20); ←
      System.out.println("Sum of 10+20 = "+result);
   }
}
```